An AI System for Large Open Virtual World

Tomáš Plch[†‡] and **Matěj Marko**[†] and **Petr Ondráček**[†]
[†]Warhorse Studios, Pernerova 53, Prague, Czech Republic
{tomas.plch,MattEntrichel}@gmail.com, petr.ondracek@warhorsestudios.cz

Martin Černý[‡] and **Jakub Gemrot**[‡] and **Cyril Brom**[‡]
[‡]Charles University in Prague, Faculty of Mathematics and Physics
Malostranské náměstí 25, Prague 1, Czech Republic
{cerny.m,jakub.gemrot}@gmail.com, brom@ksvi.mff.cuni.cz

Abstract

In recent years, computer games have reached unprecedented level of graphical fidelity to the real world. As the non-player characters (NPCs) in the game world look more and more realistic, players expect them to manifest believable behavior as well. This is accented especially in games that feature large open worlds, which players may explore freely and it is thus not possible to explicitly account for all possible player interactions. In this paper we focus mainly on *ambient AI* — the logic behind day to day behaviors of NPCs as they sleep, work and entertain themselves in the virtual world. In this context, it is of great importance to build a system that handles many NPCs (up to several hundreds) quickly. In this paper we report on an implementation of a particular AI system that was approved for deployment in an upcoming high-budget game. The system features a hierarchy of control similar to the subsumption architecture and a visual agent-based language inspired by behavior trees. We describe the challenges involved in building such a system and specific design decisions we have made that let us achieve a level of behavioral fidelity unmatched by existing games. Finally we evaluate the performance of the system in a realistic setting.

Computer games are a very specific AI application area. A particularly interesting subclass of games are those featuring a large 3D world that is open (the player may roam freely through the environment) and inhabited by a plenty of non-player characters (NPCs). The interaction with the NPCs is often at the core of the gameplay and demands a reasonable NPC AI.

In a typical open world game, such as Grand Theft Auto (Rockstar Games 2013) or The Elder Scrolls: Skyrim (Bethesda Game Studios 2011), the NPC AI may be divided into several main components, although some of the components may not be present in a particular game. As fighting enemies is still a major part of most contemporary games, *combat AI* is often the largest AI subsystem. It may be further divided into *enemy AI* that guides NPCs opposing the player and *ally AI* that controls NPCs trying to help the player in a fight. *Non-combat AI* governs the rest of the NPC behavior. It may be further divided into direct interactions with the player (e.g., dialogues, barter, . . .) and *ambient AI* which covers the daily life of the NPCs and other actions they perform on their own. It is the ambient AI that makes the world appear alive to the player. In a particular game the individual components may be further subdivided and other components may be added to suit the needs of the game (e.g. a component that coordinates groups of NPCs).

Enemy AI and direct interactions with the player are well managed in contemporary games. The support of meaningful ally AI is more problematic, as the NPC is required to be helpful to the player without being able to see "into his head". Nevertheless, multiple games have tackled this issue with results that were workable, if not satisfactory.

Ambient AI appears to be the least developed of the aforementioned components. Contemporary games have very limited support for ambient AI — in particular, almost all high-budget commercial games do not actually simulate NPC behaviors outside the area directly surrounding the current player's location. This leads to various disturbances of the believability of the virtual world when the player reenters a place he has already visited. The world state is either completely conserved, which is implausible if the player was away for a longer time, or randomly regenerated, which is implausible if the player was away for just a few seconds.

In this paper we present an AI system for an upcoming high-budget open world RPG game that allowed us to create and manage a new level of ambient AI. The system consists of multiple components connected in a manner similar to the subsumption architecture (Brooks 1991). Most of the components use a visual agent-based language to express behavior logic. The system also supports level of detail (LOD) AI (Brom, Šerý, and Poch 2007).

The rest of the paper is organized as follows: first the issues that arise during ambient AI development are discussed along with related work. Then the details of our system are presented along with results of preliminary evaluation.

The Issues

What prevents games from having a more complex ambient AI? There are three main reasons: performance, complexity of transitions and financial budget. The main cause of the performance issues are the high demands of graphical rendering and physical simulation. Basically, the graphical computations easily consume as much computing time as

they are allowed to and still ask for a little more to make that one shadow look better. In this context, only little time is left for AI. As the number of NPCs exceeds a few dozens, even running A* regularly to find paths for the NPCs becomes a serious performance issue.

Complexity of transitions is a different problem. As mentioned above, the NPC AI can be divided into components which have different functions and may use different deliberation mechanisms. It is thus necessary to regularly switch between the individual components. The transitions involved in switching between the components occur at several levels. The most important are the animation, speech and behavioral transitions. Although requirements for behavioral fidelity are not very high, cinema-like quality of animations and speech is expected. For this reasons, the transitions are not trivial, as they need to look smooth and meaningful: at minimum, transitional animations must be performed (e.g. standing up for combat if the NPC was sitting) and speech must be correctly terminated. Further technical issues are involved such as handling items that are attached to NPC's hands. Simple ambient AI needs only simple transitions and thus is preferred in present-day games.

To create a large open world, many different behaviors need to be developed. The time required to create the behaviors directly manifests in high budget requirements for AI which are often simply not fulfillable. Cutting down the costs of AI development is thus an important part of practical game AI application. Therefore it is vital to have good development support and facilitate code reuse, modification and debugging. From this perspective, AI is not only algorithms, but also software engineering and tooling support. Our previous research has shown that available tools have significant impact on the success of an AI technology (Gemrot, Hlávka, and Brom 2012).

Related Work

Many high-budget open world games such as the very successful GTA V (Rockstar Games 2013) only spawn non-story NPCs once they become close to the player, let them wander to a random location and destroy them once they get further away. Other games, such as Dragon Age (BioWare 2011), have NPCs confined to a single place for the whole game and the NPCs simply loop one or several animations at that place.

The best (in terms of player experience) open world ambient AI we have met is present in Elder Scrolls V: Skyrim (Bethesda Game Studios 2011). As we are not aware of any official description of the Skyrim AI system, our assumptions are based on our own reverse-analysis[1]. NPCs in Skyrim divide the day into several timezones (morning, day, evening, night). For each time zone they have several places they visit. An NPC chooses one of the places and stays there for a certain time, sequencing several possible animations. Some of the behaviors assigned at one place may even involve movement to a quite distant place carrying goods or a short conversation with another NPC. The selection of the behaviors however seems to be completely random and the actions have no lasting effect on the environment. The NPCs are simulated only in direct proximity of the player and randomly repositioned when player returns to the place. Although the underlying logic is simple, the palette of the behaviors the NPCs perform is relatively large. This results in a good illusion of living world as long as the player does not stay long at one place.

The S.T.A.L.K.E.R series on the other hand involves continual simulation of all NPCs in the game world, including fighting other NPCs and monsters and limited ambient behavior (moving between locations and playing animations at predefined places) (Iassenev and Champandard 2008). S.T.A.L.K.E.R initially aimed for more complex ambient behaviors, but due to the arising complexity and interactions with the game's story line, the development team decided to stick with simpler ambient AI. The ambient AI also has a narrow palette of behaviors.

Compared to Skyrim our system aims at creating more believable experience — simulating the whole world for the whole time and adding purpose and a bit more order to the behaviors of the NPCs. In respect to S.T.A.L.K.E.R we want to enforce stricter designer control and to enrich the ambient behaviors with regular daily routines and interactions with objects in the world (e.g. taking and using tools).

The System

There were several use cases we had in mind while designing the system. We needed fluent and intelligent combat behavior and interactions with the player, fully controlled by game designers, i.e. programmable in some scripting environment. Moreover, we aimed for complex ambient AI — for example a fully simulated pub. In such a pub, customers enter and leave as they wish, choose a place to sit, order beer or food, have their request heard and handled by the innkeeper who brings them the desired goods, which are in turn consumed. Beer consummation should result in change of NPCs' behavior (intoxication). The pub is fully simulated to seamlessly handle changes in the game world and interactions between the player and the NPCs. E.g., when the player prevents the innkeeper from handling guest requests in a timely manner, they complain and leave the pub.

While the pub scenario might seem simple, it is to be noted that to achieve cinema-like quality there are lots of details to deal with. Especially it is necessary to precisely synchronize animations so that the innkeeper does not put beer on the table through a NPCs head and NPCs are correctly aligned for sitting on a chair. Moreover, any of the behaviors may be interrupted when a high priority event must be handled (e.g., the NPC is attacked, or a player wants to talk to it).

Due to financial and human resources constraints, the people actually creating the behaviors in game industry — the *scripters* — are not fully qualified programmers. Scripters usually have general computer science background and minor programming knowledge but they are not always effective at creating and debugging complex code. Thus scripters need to be equipped with good higher level tools that would

[1] Our notes on ambient AI in contemporary games may be found online at `http://popelka.ms.mff.cuni.cz/~cerny/AIOpenWorlds.pdf`

let them focus on expressing the behavior logic instead of the syntax and technical details of the system.

Since the system runs inside a game-engine, it needs to adhere to the engine's execution model. The engine provides control to subsystems (e.g. renderer, AI engine, physics simulator, network interface etc.) and relies on receiving the control as soon as possible to pass it on to the next subsystem. The individual executions of the subsystems (called *updates*) must be fast enough for the engine to maintain illusion of continuous movement of the scene. To get the best of the player's hardware, new update is performed immediately after the previous one has finished and the subsystems update their internals based on the *delta* — how much time has passed since the last update — e.g., the animation system interpolates the drawn animation. Generally at least 30 updates per second must be performed for the game to be considered to run smoothly.

This cooperative multitasking design is used because it avoids synchronization which is computationally costly. This architecture is generally kept in multiprocessor environment. In this case some of the lengthy tasks are scheduled to their own threads and updates to some subsystems are run in parallel.

As the available computing time is very limited (our system was required to spend at most 5ms per update using a single processor core), the NPC action selection mechanism must feature some kind of reactive planning; more complex reasoning is not acceptable.

The AI system must respect the interface the game engine provides to the world. In game engine, there are no high level actions. The only available actions are movement and playing animations. The AI system may also directly alter the world state (create / destroy objects, etc.). For example, picking up an item is composed of movement to a desired place, playing a picking animation and removing the taken object from the game world and adding it to the inventory. The animation must be precisely positioned for the NPC's hand to actually touch the item during its movement.

Hierarchy of Control

In our system, every NPC has a *brain*. The purpose of the brain is to distribute updates to individual AI components, which we call *subbrains*. The brain also decides which subbrain may become active — multiple subbrains may run in parallel, as long as they do not conflict (e.g. during walking command issued by one specific subbrain, another subbrain may trigger a waving animation for the hands of the NPC). As of now, there is a combat subbrain, an ambient AI subbrain, a subbrain running our situation system (Cerny et al. 2014a) and specific subbrains for non-NPC AI entities. We further envision a quest subbrain that becomes active if the NPC got a role in a quest, a subbrain for reacting to non-combat high-priority events (crime, fire, . . .) and a subbrain for dialogues and other non-combat interactions with the player.

The subbrains are ordered by priority and a hierarchy of control similar to the subsumption architecture (Brooks 1991) is used. Unlike subsumption architecture, subbrains are not interrupted immediately when a higher-priority subbrain tries to run. Instead, the interrupted subbrain cooperates on the transition behavior and must terminate its operation before the interrupting subbrain is allowed to run.

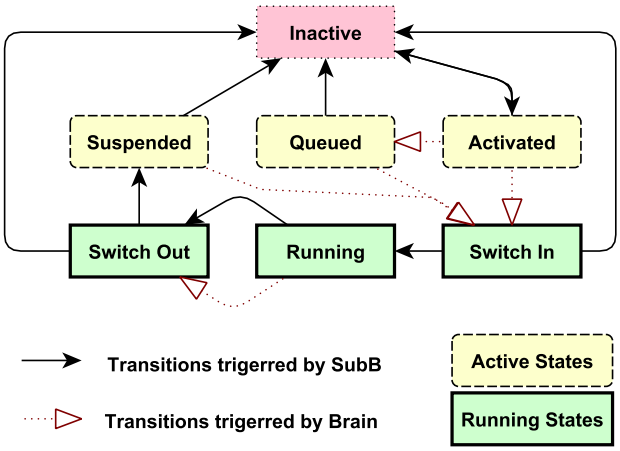

Figure 1: Graph of subbrain state transitions.

We have not implemented non-cooperative stopping of subbrains as this may introduce huge issues in the context of a game engine. The most serious trouble arise from the interaction of physics and character animations. Improper termination of animation may lead to undesirable consequences including the NPC being stuck "inside a table" or ejecting the NPC high above the ground.

There are three groups of states the subbrain may take: *inactive*, *active* and *running*. In running states, the subbrain is in control of the NPC. Active states signal that the subbrain wants to enter a running state and control the NPC. In the inactive state, the subbrain does not compete for control of the NPC, it only checks whether conditions for its activation are met. Transitions between inactive and active states are in full control of the subbrain while the brain controls the transitions to running states. The complete transition graph of subbrain states is given in Figure 1. This system is inspired by previous academic research (Plch 2009).

When a subbrain is interrupted by a higher priority subbrain it may either stop completely or it may become suspended. Suspended subbrain retains its internal state and is scheduled for execution as soon as all of the higher priority subbrains cease to be active. There is also mandatory "Switch In" and "Switch Out" phase of execution to ensure proper initialization and cleanup. If the subbrains cooperate on a switch (i.e. there is a special transition behavior between the two subbrains), the Switch Out phase runs in parallel to Switch In phase of the newly active subbrain.

By entering the "Activated" state, the subbrain signals to the brain that it wants to compete for execution. If no subbrain is running, the brain lets it run directly. If the running subbrain is of lower priority, the activated subbrain is transitioned to the "Queued" state which signalizes to the subbrain that it is scheduled for execution once the running subbrain switches out.

Modular Behavior Trees

For easy development of behaviors, our system features a visual agent-based language inspired by behavior trees (Champandard 2007b). We call this system *modular behavior trees* (MBTs)[2]. In plain behavior trees, the behavior code consists of a tree. The leaves of the tree are atomic actions and senses while the internal nodes (called *composites*) represent structure and decision logic of the behavior. Evaluation of a node may return three possible values: `success`, `failure` and `running`.

Actions return `success` when the NPC has finished the action, `running` if more is to be done and `failure` if the action cannot complete. Senses evaluate a condition in the world and succeed if it is true and fail otherwise. Composites are either *selectors* or *sequences*; both evaluate their children in order and when the evaluated child returns `running` , they also return `running`. Selectors return `success` when the first child node succeeds and do not evaluate the rest of the children. Sequences on the other hand need all of their children to succeed in order to return `success`. To perform more complex logic or computations, some kind of scripting language may be invoked by tree nodes; in our case we used LUA (Schuytema and Manyen 2005).

This simple formalism allows for easy coding of quite complex behaviors and variations of behavior trees have become a de facto industry standard. The actions and senses are directly implemented by programmers in the game engine and thus are quick to evaluate. Another advantage is that subtrees may be easily reused among different behaviors. Similar reactive planning approaches have been previously evaluated in academia (Bryson 2001). Further extensions to the formalism including *decorator* nodes altering the execution context (Champandard 2007c) of the subtree and parallel execution (Champandard 2007a) were proposed.

We have however identified downsides to the basic idea and improved both the syntax and the semantics of the trees to better express complex behaviors. The key issues were the node execution model, limited variable support, missing synchronization and communication mechanisms, no explicit time awareness and tool support.

Node Execution. In a plain behavior tree, the whole execution is stateless and the senses that guard the individual tasks are continually reevaluated. This introduces high reactivity to external stimuli, but it may be computationally intensive. In some other implementations, the composites have internal state and continue evaluating their children starting at the first one that returned `running` in the previous iteration. However, the reactivity may be greatly reduced this way.

We have decided to allow for more flexibility and let the individual node decide what state should be kept and how children are evaluated and how their state influences the state of the composite. In this way, it is the designer who chooses, which decisions need to be reactive and which should maintain state. Apart from stateful and stateless variants of selectors and sequences, this allowed us to introduce more types of composites to the language, many of them corresponding to constructs of classical programming languages.

Most notably we introduced loops, time-limited execution, parallel execution of multiple subtrees, "calls" of external subtrees and *decorators* that alter the result returned by a subtree. We have even introduced support for creating finite state machines (Fu and Houlette-Stottler 2004) inside the trees. While state machines are difficult to maintain for complex behaviors, they are very good at expressing simple stateful behaviors. By including state machines at the lowest level of the MBTs we are able to take the best of both approaches.

Since the composites have full control over execution, they may return control before an actual action is issued (e.g., if the time budget for decision making has run out). This further facilitates performance guarantees of the system and the locality of decisions allows the nodes to easily maintain consistency and have clear semantics. The execution semantics also allow for good debugging support. In particular, breakpoints may be attached to various transitions of the node state or to individual updates.

A node starts in the *none* state. Prior to execution, there is a mandatory initialization phase which has to be atomic and immediate. Succesful initialization transitions the node to the *initialized* state, if the node cannot be executed for some reason, it transitions to the *failure* state instead.

When node starts performing the actual work, it transitions to the *running* state. Once the work is done — which may be in the same update as the work has started — the node transitions to either *success* or *failure* state. Before the node is executed again, a mandatory atomic cleanup is performed, transitioning the node back to the *none* state.

As nodes have state, there are two ways to interrupt node execution. The node could either become *suspended* and retain its state, or *halted* and clear its state. However, as was already mentioned, some behaviors may not be interrupted abruptly. Thus the MBT allows for intermediate states (*halting*, *suspending*) that signal that the node is being halted or suspended, but still needs further updates. In a symmetrical fashion, prior the node leaves the *suspended* state, it passes through an intermediate *resuming* state that lets the node to prepare for further execution and may also last for multiple updates. Once the node enters *resumed* state, it is transitioned back to the *running* state. Since the nodes are in full control of the execution, we were able to create decorator nodes that evaluate a subtree that cleans up once the child has been suspended or halted (e.g., drops items the NPC has held in hands) or a subtree that checks consistency after resuming. This further increases designer control over the behaviors and eases maintaining behavioral consistency. The complete graph of possible node states and transitions is given in Figure 2.

Subbrains and MBTs. Most of our subbrains use MBTs while in running states, but the subbrains for various AI components are not technically the same. They differ in conditions for activation, frequency of updates and the way they

[2]MBTs have been briefly introduced in our previous work (Plch et al. 2014), here we present more details of their inner workings.

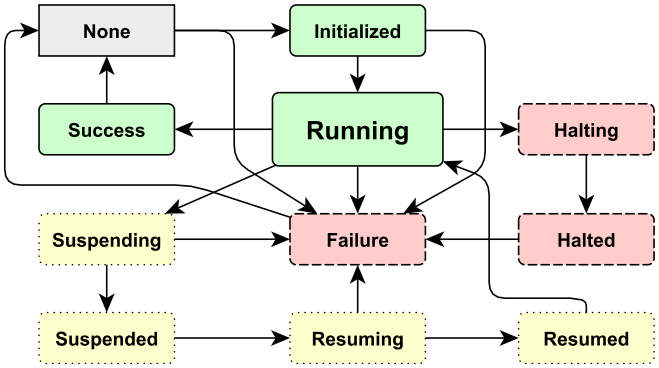

Figure 2: States of the node execution. The appearance of the states is only a visual aid to highlight the most important transition sequences.

obtain the tree to execute.

The execution of the root MBT node and its updates are controlled by the subbrain. In particular, the subbrain decides whether switching out the subbrain should result in suspending or halting the MBT.

While subbrain switching bears resemblance to MBT node execution, the subbrain system could not be modeled by single larger MBT, because some of the subbrains do not use MBT at all or wrap them in special logic that could not be easily expressed with MBTs. Moreover it maintains a clear separation between AI components and lets different scripters focus on different aspects of AI behavior.

Types and Variables. In most engines, the data accessible from the NPCs behavior tree is limited to either a set of hard-coded states or values provided by the engine (e.g. a boolean InDanger that indicates that the NPC faces a serious threat) or only information in a simple "key — value" pairing. In order to create a versatile data model we have introduced a simple type system.

Every type definition is similar to a struct construct of the C language. The individual members are either primitive types (boolean, integer, float, string, and *any* representing any possible type) or types defined previously. It is also possible to create a new type by extending an already defined type and adding new members. Both indexed and associative arrays are supported but nesting of arrays is prohibited to keep the code clean and running fast.

The MBT may define unlimited number of variables which may be substituted for any parameter of a node. Basic arithmetical expressions are evaluated.

Synchronization and Communication. To create complex behaviors, synchronization and communication between NPCs or between two branches of the same tree (in case of parallel execution) are necessary and support for both was added in MBTs. Messages are simply data of a predefined type sent from one NPC to one or multiple other NPCs. Each NPC has a list of associated *inboxes*, each inbox has a type of data it receives, priority and possibly further filtering logic. For each inbox (ordered by priority) type compatibility is checked with the type of the message and filtering rules are applied. If the message matches the inbox, it is put inside the message queue in the inbox.

Special MBT nodes are introduced to send messages with various types of target selection and for both blocking and non-blocking reading of messages from specified inbox into a tree variable.

Since the messaging system is synchronized among NPCs, it would be, in theory, sufficient to implement NPC synchronization. This however would not be very practical. Since the most common synchronization task in the game is the need for multiple NPCs to start a task at the same moment, we have introduced special nodes to handle just that. The nodes block at shared semaphores until a specified number of NPCs subscribe to the semaphore. The execution model of the nodes allows for consistent acquire and release of the semaphore in response to suspending or halting the node. Thus the NPC may easily wait for the lock in a parallely executed subtree of the behavior and perform meaningful actions while waiting.

Time Awareness. In plain behavior trees, time is considered only at the action level (e.g. to correctly update NPC animation) but not in the internal nodes. Most notably, at most one action of a sequence is performed in every update. This leads to more complicated design if a game requires consistency of execution even when a LOD policy is in place and the NPC is updated scarcely; e.g., if an NPC playing cards (and cheating) should win a coin every minute, but is updated in 3 minute intervals, it should receive 3 coins in each update. Note however, that updates may be scarce only if the NPC is not visible to the player, otherwise animation and movement actions could not be performed instantaneously without the player noticing glitches.

To resolve this issue, the MBT tracks the interval since its last update (called *delta*) and action nodes estimate the game time they will need to finish. If this estimated time is less than delta, the effects of the action are performed immediately and the estimated time is subtracted from delta. The execution of the MBT than continues until the delta is reduced to zero, i.e. until an action consumes the rest of the delta without finishing.

If there is not enough computing time to evaluate nodes that would consume the whole delta, the delta is kept and added to the next update. This is also useful if a multi-step computation is taking place inside the tree (e. g., iterating over neighbouring NPCs to find a good action target) — multiple steps may be performed in one update, if computing time is available. This way varying time flow rate for different NPCs is possible and upper bounds on MBT evaluation time may be imposed. Example time flow is given in Figure 3.

The Tools. Our previous research has indicated that good tool support is vital for success of a technology (Gemrot, Hlávka, and Brom 2012). We have created a visual editor for MBTs with drag and drop support and direct editing of node parameters. This way scripters need not worry about the tree syntax and search for names of parameters. The editor also highlights execution states of the nodes by different colors to ease debugging and displays current values of all variables.

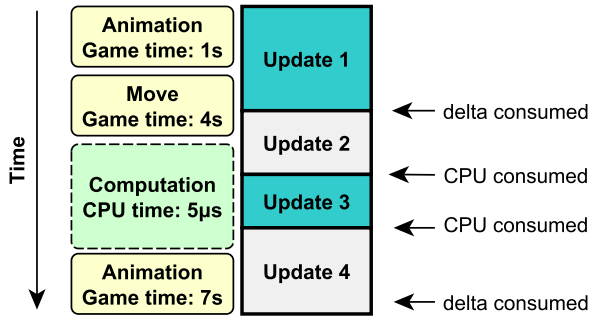

Figure 3: Example of time flow during node execution. The NPC is updated only scarcely with delta $= 3s$. 2 µs CPU time limit is imposed on every update. For simplicity, movement and animation is considered to consume 0 CPU time. In Update 1 first action is finished and second one is started, because delta is large enough. In Update 2 and 3 the CPU budget is exceeded due to a lengthy computation and the unconsumed delta is stored. Thus in Update 4, the total delta is 7s, letting the last action to finish immediately.

A breakpoint may be added to a node for various phases of its execution. When a breakpoint is hit, the whole virtual environment is paused and the user may explore states of all NPCs in the game including values of all variables.

Evaluation

In our evaluation we focused on the MBT system which is crucial for good standalone ambient AI. We have performed two types of evaluation with MBTs: qualitative evaluation where scripters tried to implement the same scenario with different tools and a preliminary quantitative evaluation of the speed of the whole system. We have also tested the subbrain switching which, in combination with the cleanup mechanisms inside MBTs has proved powerful enough for seamless switching between AI components.

Qualitative Evaluation of MBTs

We envisioned two basic scenarios we considered adequate for testing MBTs. Two of our scripters implemented them with MBTs in various stages of the development. They also tried to implement the scenarios with two previously deployed technologies within our game engine. The first one was plain behavior trees with conditions evaluating only boolean variables and allowing for finite state machines as leaves of the tree (BT1), the second one was behavior trees with boolean based conditions (i.e. no relational operations) retaining node states (BT2). Both technologies had support for communicating with LUA scripts.

Pub. The first tested scenario was a pub. The pub is an area, where NPC can come in and sit at a free table. There is no guarantee about how many and which seats are available and how many and which NPCs will arrive to the pub at any given time. The pub has two inhabitants: an innkeeper and a waitress. If a customer NPC comes into the pub and sits down at a table, orders a beer, drinks it and then either continues with another beer or leaves. The innkeeper drafts the beer from the tap while the waitress serves them to the guests. The waitress reduces the walking distance by handling the customers who are close together in one go, ignoring in part the order in which requests were made. If an NPC does not receive its order in a reasonable time, it will get angry and leave the pub. Finally if a special NPC arrives (a rich and valued guest), it is served as quickly as possible.

The solution for this scenario builds upon the idea of a "governor" of the area. In first iterations the innkeeper was the governor, but later we switched to governor being a separate disembodied entity communicating with both guests and the innkeeper. When an NPC enters the pub, it asks the governor (through a message) for a free place to sit. If it orders a beer, it plays a waving animation and sends a message to the governor. The governor keeps the waitress informed about the incoming orders and available beers to deliver. The waitress internally reorders the requests and delivers the beers to tables to minimize the travel distance. Important advantage of this solution is that only the governor needs to know where the places to sit are.

Our scripters tried to model this in BT1 but they did not succeed. With BT1 the code got complicated and large and the absence of variables prevented some of the features to be achieved at all. It finally went to a point, where most of the logic was shifted from BT1 into LUA scripts which was hard to debug and maintain while the NPCs were rigid and predictable. Utilizing BT2 has led to better looking behaviors but with similar issues. All NPCs were largely written in LUA, shared almost all their internal variables in a global LUA table, leading to a rather unreadable and unsafe code.

Our architecture allowed scripters to utilize variables for specifying internal NPC data, such as the amount of ordered beers, the time spent waiting for the waitress, the currently closest table etc. The message system was utilized as a natural communication medium between the NPCs. The typed messages allowed to encode information like "who is requesting a seat" and "how long am I waiting" etc. without any hard-coded enums or preset data. The parallelization of execution was used to receive and evaluate messages while executing animations to make the scene more life-like. We also noticed that while with the prior system the frame rate dropped considerably with more than about 8 NPCs present, our system managed to scale properly.

Shop. Our second scenario was a shop. NPCs visit the shop where they have to stand in a queue and are one by one served by the shop keeper. A NPC can run out of patience and leave the shop earlier. A key notion here is that the code should not have hard-coded maximal length of the available queue, so a small shop could have the same code as a large one. It is required that the NPCs in the queue move to the next position if NPC at any point leaves the queue. The shopkeeper was designated the governing entity in the shop and managed the queue.

The shop scenario proved even harder to manage with both BT1 and BT2, since it was not possible to encode the varying length of the queue and advances if someone had left the queue. Once again, the behaviors were almost com-

Table 1: The results of the quantitative evaluation. The table displays mean and .99 quantile / maximum (in brackets) times taken by individual components of the AI system. All times in ms per frame.

Scenario	NPCs	MBTs		Brain		Sensation		Other		Total	
simple	100	0.33	(0.6 / 1.7)	0.09	(0.2 / 0.5)	0.02	(0.2 / 0.4)	0.34	(0.6 / 1.7)	**0.79**	**(1.3 / 3.9)**
	200	0.53	(0.9 / 2.5)	0.20	(0.3 / 0.5)	0.03	(0.3 / 0.7)	0.42	(0.8 / 1.8)	**1.19**	**(2.0 / 4.3)**
	300	0.75	(1.1 / 4.0)	0.30	(0.5 / 3.3)	0.05	(0.4 / 1.1)	0.59	(1.0 / 4.0)	**1.71**	**(2.6 / 6.8)**
daycycles	10	0.39	(0.7 / 1.4)	0.12	(0.2 / 0.6)	0.02	(0.1 / 0.5)	0.22	(0.5 / 1.0)	**0.76**	**(1.1 / 2.0)**
	20	0.46	(0.7 / 1.6)	0.12	(0.2 / 0.4)	0.02	(0.6 / 2.8)	0.22	(0.6 / 2.8)	**0.84**	**(1.4 / 3.4)**
	30	0.56	(0.9 / 1.8)	0.14	(0.2 / 0.5)	0.02	(0.1 / 0.3)	0.23	(0.7 / 2.7)	**0.96**	**(1.6 / 3.9)**

pletely specified in a hard-to-manage lengthy LUA code.

Our architecture managed well, since the code for the shopkeeper to keep track of the queue could be specified without LUA. When the queue advanced, the shopkeeper sent messages to the affected NPCs. The other NPCs simply waited for the messages and notified the shopkeeper whenever they entered or left the shop.

Both our scenarios show where the limits of plain behavior trees without any typed variables or message passing lie. Our architecture was capable of solving the issues robustly, since adding a table in the pub or changing the size of the shop required only change of data but not of the actual behavior code. In contrast, code created with the other technologies was too specifically tailored to the given task and had to be rewritten if scenario conditions changed. Moreover the scripters considered MBTs relatively easy to learn and did not have trouble understanding its semantics. They were also very fond of the debugging features which were superior to the other systems as well as to the LUA implementation we use.

In the end we started instructing our scripters to use LUA as little as possible for two reasons: first, invoking LUA environment was computationally expensive and second, LUA code was less readable to the scripters than the visual structure of MBTs. On the other hand LUA nodes proved very useful as a tool to prototype new functionality that will later be added as a specialized MBT node.

Quantitative Evaluation of MBTs

In first set of scenarios (simple) there were many NPCs with a simple tree (10 nodes, depth 4). The NPCs moved to random positions at various speeds, while the tree was enlarged by spurious decorators and composites. The second set of scenarios (daycycles) were production ones — lower number of NPCs carrying out their daily routines, including hoeing fields, visiting pub and eating (trees with $60+$ nodes and maximal depth > 10). Aside from the NPCs, the environment in the daycycles scenarios contained also 142 non-NPC entities (so called smart objects and smart areas) which are primarily repositories of behaviors the NPCs use but a smaller part of them also has some internal logic (expressed by MBTs) to coordinate NPCs with a specific behavior (similar to the governor in the pub scenario).

Both scenario sets were tested with different number of NPCs, see Table 1 for the results. All NPCs were updated every frame (no LOD). To keep the results meaningful, no CPU budgeting restrictions were enforced — the trees always run until an action was executed. Data was gathered running the game for 3 minutes, resulting in 3000 - 6650 captured frames. To reduce noise caused by interrupts from the operating system or other processes, up to 10 outlying frames were removed from each measured category.

The project leads gave us a budget of 5ms per frame (on a single core) for the whole AI system. Even with plenty NPCs on the scene, the average and the .99 quantile performance is far below the limit, although the peak performance is not satisfactory. But as there are high peaks in MBT evaluation, enforcing CPU budget restrictions should effectively cut the peaks, postponing some of the workload to next frame. Since at maximum 1 in 100 frames is over the limit, this will not have any detrimental effect on the resulting behavior. Proper use of LOD to limit the number of frequently updated NPCs should be enough to keep the system within bounds in the envisioned production load.

Conclusions

We have presented an overall structure of an AI system for a large open world RPG game currently under development. The system has been shown to handle the intrinsic complexity of NPC behavior well without sacrificing performance. The system has been approved by project leads for deployment in the game and has replaced the system shipped with the game engine. The flexibility of the system allowed us to extend the AI with further components, such as smart areas (Cerny et al. 2014b) to further mitigate complexity and increase modularity.

As a future work we plan to extend the system with elements of virtual storytelling and experiment with adversarial search for real-time decisions within the combat subbrain.

Acknowledgements

This research is partially supported by the Czech Science Foundation under the contract P103/10/1287 (GAČR), by student grants GA UK No. 559813/2013/A-INF/MFF and 655012/2012/A-INF/MFF and partially supported by SVV project number 260 104.

Special thanks belong to Warhorse Studios and its director Martin Klíma for making this research possible by their openness to novel approaches and by letting researchers work in close cooperation with the company.

References

Bethesda Game Studios. 2011. Elder Scrolls V: Skyrim. http://www.elderscrolls.com/. Last checked: 2014-02-24.

BioWare. 2011. Dragon Age II. http://www.dragonage.com/. Last checked: 2014-02-24.

Brom, C.; Šerý, O.; and Poch, T. 2007. Simulation level of detail for virtual humans. In *Intelligent Virtual Agents*, 1–14. Springer. LNCS 4722.

Brooks, R. A. 1991. Intelligence without representation. *Artificial Intelligence* 47:139–159.

Bryson, J. 2001. *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents.* Ph.D. Dissertation, Massachusetts Institute of Technology.

Cerny, M.; Brom, C.; Bartak, R.; and Antos, M. 2014a. Spice it up! Enriching open world NPC simulation using constraint satisfaction. In *Proceedings of Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. This proceedings.

Cerny, M.; Plch, T.; Marko, M.; Ondracek, P.; and Brom, C. 2014b. Smart areas: A modular approach to simulation of daily life in an open world video game. In *Proceedings of 6th International Conference on Agents and Artificial Intelligence*, 703–708.

Champandard, A. 2007a. Enabling concurrency in your behavior hierarchy. *AIGameDev.com*. http://aigamedev.com/open/article/parallel/ Last checked 2014-02-28.

Champandard, A. 2007b. Understanding behavior trees. *AIGameDev.com*. http://aigamedev.com/open/article/bt-overview/ Last checked 2014-01-05.

Champandard, A. 2007c. Using decorators to improve behaviors. *AIGameDev.com*. http://aigamedev.com/open/article/decorator/ Last checked 2014-02-28.

Fu, D., and Houlette-Stottler, R. 2004. The ultimate guide to FSMs in games. In *AI Game Programming Wisdom II*. Charles River Media. 283–302.

Gemrot, J.; Hlávka, Z.; and Brom, C. 2012. Does high-level behavior specification tool make production of virtual agent behaviors better? In *Proceedings of the First international conference on Cognitive Agents for Virtual Environments*, CAVE'12, 167–183. Berlin, Heidelberg: Springer-Verlag. LNCS 7764.

Iassenev, D., and Champandard, A. 2008. A-Life, emergent AI and S.T.A.L.K.E.R. *AIGameDev.com*. http://aigamedev.com/open/interviews/stalker-alife/ Last checked 2014-02-24.

Plch, T.; Marko, M.; Ondracek, P.; Cerny, M.; Gemrot, J.; and Brom, C. 2014. Modular behavior trees: Lanuage for fast ai in open-world video games. In *The proceedings of 21st European Conference on Artificial Intelligence*. In press, 2 pages.

Plch, T. 2009. Action selection for an animat. Master's thesis, Charles University in Prague.

Rockstar Games. 2013. Grand Theft Auto V. http://www.rockstargames.com/V/. Last checked: 2014-02-24.

Schuytema, P., and Manyen, M. 2005. *Game Development with LUA*. Charles River Media.